Can Computing the Fibonacci Sequence be Parallelized?

This is a fun question. We wish to compute: \( \{F_0, F_1, F_2, ..., F_N\} \) for some fixed \( N \) chosen in advance of the computation. The Fibonacci numbers have a closed form expression:

\[
F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}
\]

where

\[
\phi = \frac{1 + \sqrt{5}}{2} \quad \text{and} \quad \psi = \frac{-1}{\phi} = 1 - \phi
\]

The derivation of equation (1) comes from the theory of sequences defined by linear recurrences with constant coefficients.\(^1\)

What’s that got to do with parallel computation?

Equation (1) can be computed independently for each value of \( n \). If we have \( N \) processors, we can compute each \( F_n \) independently.

Someone may notice that \( \phi \) is an irrational number, and we will have to use floating point arithmetic. That may lead to some rounding error, so can we still get the correct answer? For \( N \) within reason, the rounding error will certainly be less than 0.5, so we can find the integer value of \( F_n \) by rounding.

Someone may also notice that for the last number \( F_N \), we will need to compute \( \phi^N \) and \( \psi^N \). If we use the obvious algorithm, that will take \( \mathcal{O}(N) \) time, so we have no improvement over a simple sequential algorithm for \( F_N \). Fortunately, we can compute \( \phi^N \) and \( \psi^N \) in \( \mathcal{O}(\log(N)) \) time.

\[\text{Professor Fibonacci, circa 1202}\]

\(^1\)An in-depth treatment of recurrence equations and their solution is beyond the scope of our course.
**Fast Exponentiation**

The problem is to compute $x^n$ for a positive integer $n$ efficiently.

On every iteration of the main loop we successively square the input $x$. This process creates a sequence:

$$x, x^2, x^4, x^8, x^{16}, \ldots$$

At the same time, we extract the bits of $n$ from low order to high order. We include the factor of the form $x^{(2^k)}$ when the corresponding bit of $n$ is 1, and we exclude that factor if the corresponding bit of $n$ is 0.

Suppose the bits of $n$ are $b_0, b_1, b_2, \ldots, b_k$ from low order to high order. We write $n$ as:

$$n = b_0 2^0 + b_1 2^1 + b_2 2^2 + \ldots + b_k 2^k$$  \hspace{1cm} (2)

Substituting equation (2) into $x^n$ we have:

$$x^n = x^{b_0 2^0} \cdot x^{b_1 2^1} \cdot x^{b_2 2^2} \cdot \ldots \cdot x^{b_k 2^k}$$  \hspace{1cm} (3)

$$= x^{b_0} \cdot x^{b_1} \cdot x^{b_2} \cdot \ldots \cdot x^{b_k}$$  \hspace{1cm} (4)

Whenever $b_j = 1$, the corresponding factor in the product in equation (4) becomes $x^{(2^j)}$.

Whenever $b_j = 0$, the corresponding factor in the product in equation (4) becomes $x^0 = 1$.

Such factors can be ignored without changing the final result.

The complete algorithm in pseudo-code is given below:

```
Input: Number $x$ and positive integer $n$
Output: $x^n$

Method:
    \hspace{1cm} $r = 1$
    \hspace{1cm} while ( $n > 0$ ) {
        \hspace{1.5cm} if ( $n$ is an odd integer ) $r = r \cdot x$
        \hspace{1.5cm} $x = x \cdot x$
        \hspace{1.5cm} $n = n/2$
    }
    \hspace{1cm} return( $r$ )
```

In C++, we can implement the algorithm as:

```c++
1
 while ( $n > 0$ ) {
3
     if ( $n & 1$ ) $r = r \cdot x$
4
     $x = x \cdot x$
5
     $n >>= 1$
6
 return($r$);
```